Functional Programming

WS 2010/11

Christian Sternagel (VO)
Friedrich Neurauter (PS)   Ulrich Kastlunger (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

November 10, 2010

Today's Topics

- Evaluation Strategies
- Abstract Data Types
- Sets and Binary Search Trees

Evaluation Strategies

Recall - λ-Terms

$$t \stackrel{\text{def}}{=} x \mid (\lambda x.\, t) \mid (t \;\, t)$$

Recall - λ-Terms

$$t \stackrel{\text{def}}{=} x \mid (\lambda x.\, t) \mid (t\ \ t)$$

Examples

Conventions	Verbose	in Words
$x\ y$	$(x\ y)$	"x applied to y"
$\lambda x.\, x$	$(\lambda x.\, x)$	"lambda x to x" (identity function)
$\lambda xy.\, x$	$(\lambda x.\, (\lambda y.\, x))$	"lambda x y to x"
$\lambda x.\, x\ x$	$(\lambda x.\, (x\ x))$	"lambda x to x applied to x"
$(\lambda x.\, x)\ x$	$((\lambda x.\, x)\ x)$	"lambda x to x, applied to x"

Recall - β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \rightarrow_\beta t$
- iff there is context C, variable x, and terms u and v, s.t.,
- $s = C[(\lambda x.\, u)\ v]$ and $t = C[u\{x/v\}]$

Recall - β-Reduction

- term s (β-)reduces to term t in one step
- written: $s \longrightarrow_\beta t$
- iff there is context C, variable x, and terms u and v, s.t.,
- $s = C[(\lambda x.\, u)\ v]$ and $t = C[u\{x/v\}]$

Examples

$$K \stackrel{\text{def}}{=} \lambda xy.\, x$$

$$I \stackrel{\text{def}}{=} \lambda x.\, x$$

$$\Omega \stackrel{\text{def}}{=} (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$$

Order of Evaluation

- consider `d x = x + x`

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

```
d (d 2)
```

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{array}{c}
\texttt{d (d 2)} \\
\swarrow \\
\texttt{d (2+2)}
\end{array}
$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{array}{ccc}
 & \texttt{d (d 2)} & \\
\swarrow & & \searrow \\
\texttt{d (2+2)} & & \texttt{(d 2)+(d 2)}
\end{array}
$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$\begin{tikzcd}[column sep=small] & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\ \texttt{d (2+2)} \arrow[d] & & \texttt{(d 2)+(d 2)} \\ \texttt{d 4} & & \end{tikzcd}$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\
 & \texttt{d (2+2)} \arrow[dl] \arrow[d, bend left] & & \texttt{(d 2)+(d 2)} \\
\texttt{d 4} & \texttt{(2+2)+(2+2)} & &
\end{tikzcd}
$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
& & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\
& \texttt{d (2+2)} \arrow[ddl] \arrow[ddr, bend left] & & \texttt{(d 2)+(d 2)} \arrow[dll] \\
& \texttt{(2+2)+(d 2)} & & \\
\texttt{d 4} & & \texttt{(2+2)+(2+2)} &
\end{tikzcd}
$$

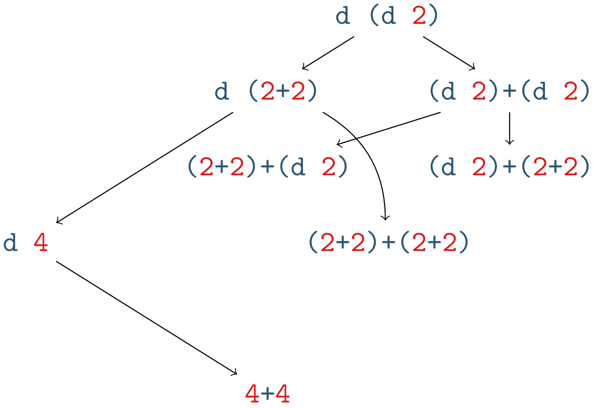
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & \\
 & \texttt{d (2+2)} \arrow[ddl] \arrow[ddr, bend left] & & \texttt{(d 2)+(d 2)} \arrow[dll] \arrow[d] \\
 & \texttt{(2+2)+(d 2)} & & \texttt{(d 2)+(2+2)} \\
\texttt{d 4} & & \texttt{(2+2)+(2+2)} &
\end{tikzcd}
$$

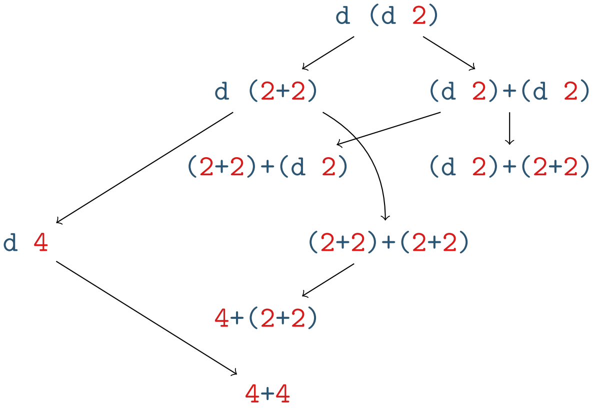
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



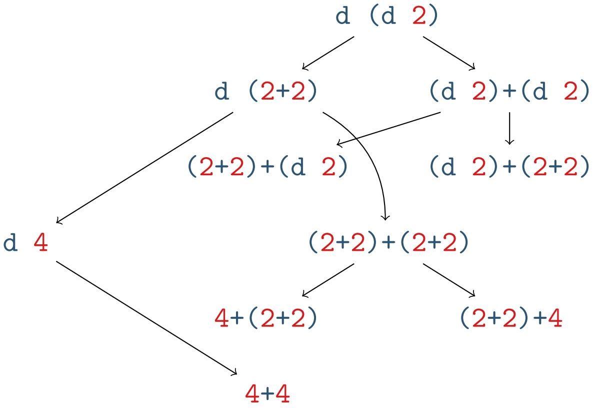
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



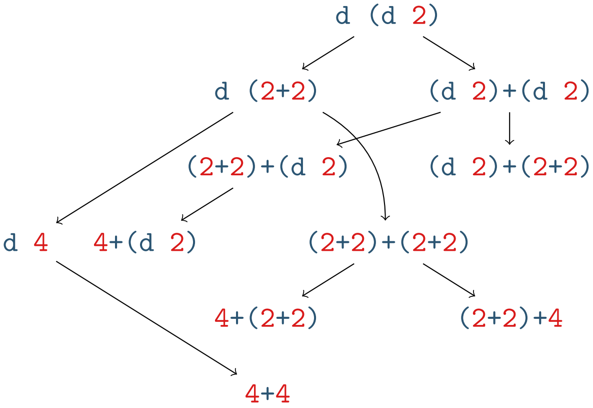
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



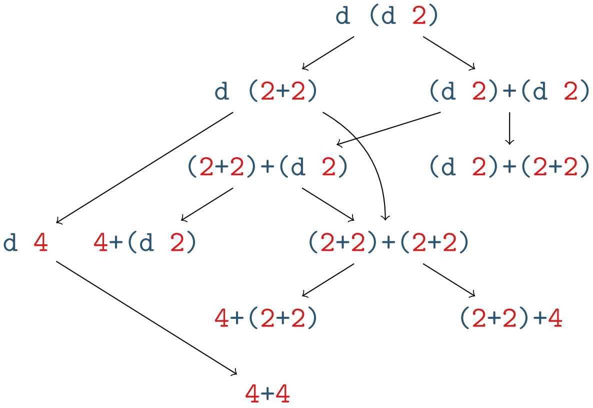
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



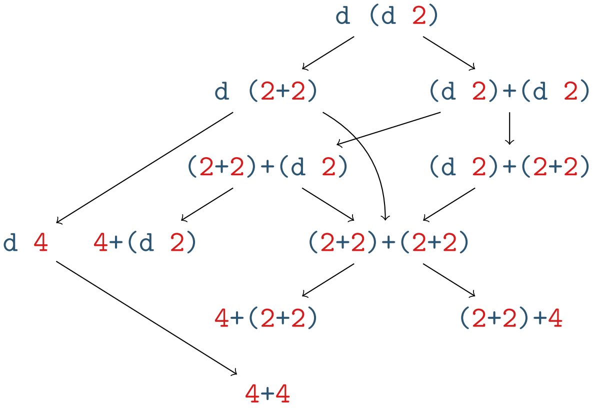
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



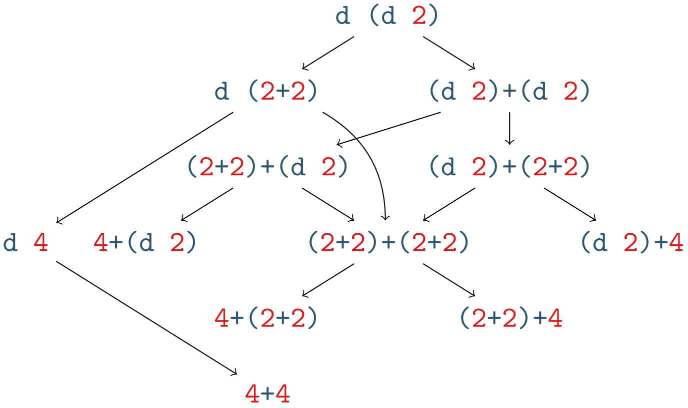
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



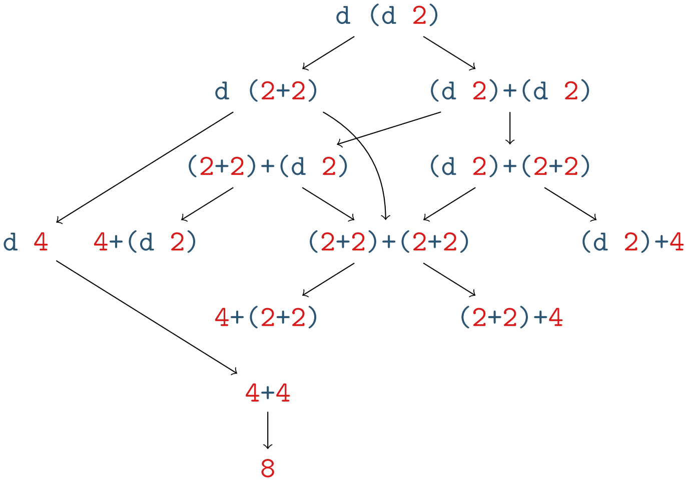
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



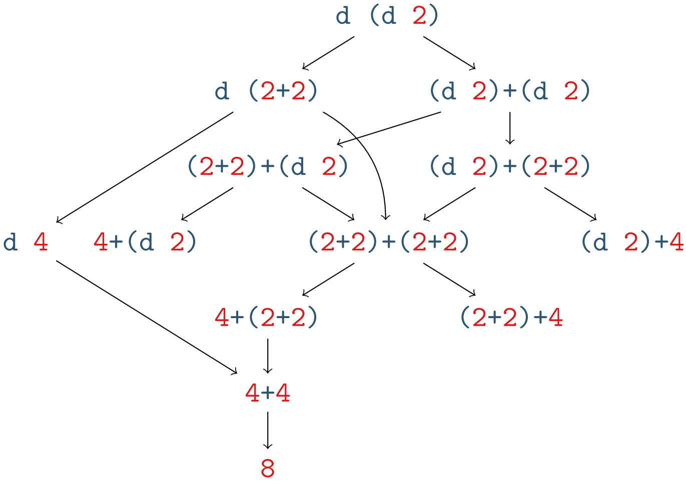
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



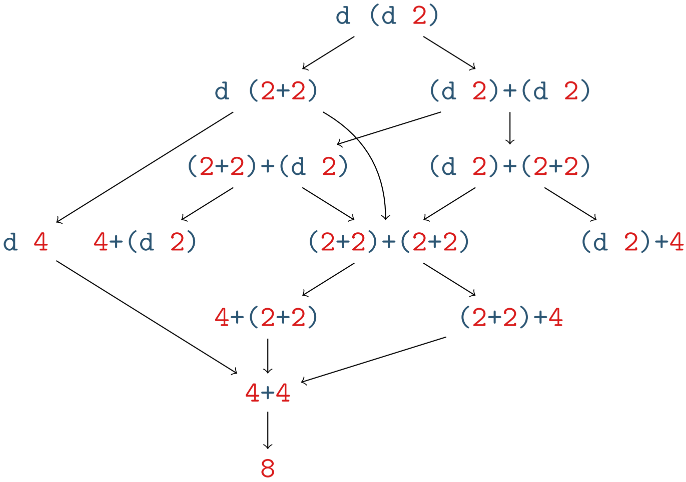
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



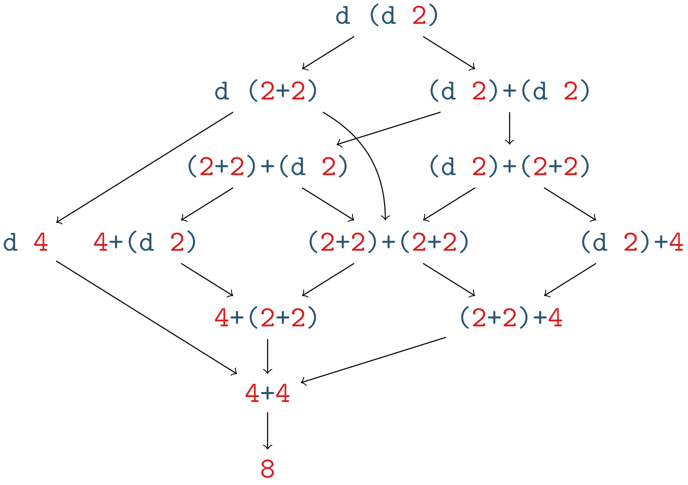
Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows

$$
\begin{tikzcd}
 & & & \texttt{d (d 2)} \arrow[dl] \arrow[dr] & & \\
 & & \texttt{d (2+2)} \arrow[ddll] \arrow[ddr, bend left] & & \texttt{(d 2)+(d 2)} \arrow[dll] \arrow[d] & \\
 & & \texttt{(2+2)+(d 2)} \arrow[dl] \arrow[dr] & & \texttt{(d 2)+(2+2)} \arrow[dl] \arrow[dr] & \\
\texttt{d 4} \arrow[ddrr] & \texttt{4+(d 2)} \arrow[dr] & & \texttt{(2+2)+(2+2)} \arrow[dl] \arrow[dr] & & \texttt{(d 2)+4} \\
 & & \texttt{4+(2+2)} \arrow[d] & & \texttt{(2+2)+4} \arrow[dll] & \\
 & & \texttt{4+4} \arrow[d] & & & \\
 & & \texttt{8} & & &
\end{tikzcd}
$$

Order of Evaluation

- consider `d x = x + x`
- the term `d (d 2)` may be evaluated as follows



Strategies

- fix evaluation order
- call-by-value (compute arguments before function calls)
- call-by-name (compute arguments "on demand")

Strategies

- fix evaluation order
- call-by-value (compute arguments before function calls)
- call-by-name (compute arguments "on demand")

Example

- call-by-value

$$
\begin{aligned}
\texttt{d (d 2)} &= \texttt{d (2+2)} \\
&= \texttt{d 4} \\
&= \texttt{4+4} \\
&= \texttt{8}
\end{aligned}
$$

- call-by-name

$$
\begin{aligned}
\texttt{d (d 2)} &= \texttt{(d 2) + (d 2)} \\
&= \texttt{(2+2) + (d 2)} \\
&= \texttt{4 + (d 2)} \\
&= \texttt{4 + (2+2)} \\
&= \texttt{4 + 4} \\
&= \texttt{8}
\end{aligned}
$$

(Leftmost) Innermost Reduction

- always reduce leftmost innermost redex
- a redex u inside a term t is innermost if it does not contain any redexes as proper subterms, i.e.,

$$\nexists C\ s.\, u = C[s],\ C \neq \Box \text{ and } s \text{ is a redex}$$

(Leftmost) Innermost Reduction

- always reduce leftmost innermost redex
- a redex u inside a term t is innermost if it does not contain any redexes as proper subterms, i.e.,

$$\nexists C\ s.\ u = C[s],\ C \neq \Box \text{ and } s \text{ is a redex}$$

Example

- consider $t = (\lambda x.\, (\lambda y.\, y)\ x)\ z$
- $(\lambda y.\, y)\ x$ is innermost redex
- t is redex, but not an innermost redex

(Leftmost) Outermost Reduction

- always reduce leftmost outermost redex
- a redex u inside a term t is outermost if it is not a proper subterm of some other redex inside t, i.e.,

$$\nexists D\ C\ s.\, t = D[s],\ s = C[u],\ C \neq \Box \text{ and } s \text{ is a redex}$$

(Leftmost) Outermost Reduction

- always reduce leftmost outermost redex
- a redex u inside a term t is outermost if it is not a proper subterm of some other redex inside t, i.e.,

$$\nexists D\ C\ s.\, t = D[s],\ s = C[u],\ C \neq \square \text{ and } s \text{ is a redex}$$

Example

- consider $t = (\lambda x.\, (\lambda y.\, y)\ x)\ z$
- t is an outermost redex
- $(\lambda y.\, y)\ x$ is redex, but not outermost redex

Exercises

- consider the λ-terms
- $S = \lambda xyz.\, x\ z\ (y\ z)$
- $K = \lambda xy.\, x$
- $I = \lambda x.\, x$
- reduce $S\ K\ I$ to NF using leftmost innermost reduction
- reduce $S\ K\ I$ to NF using leftmost outermost reduction

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation
- adopted by most programming languages
- slight modification: only reduce terms not in WHNF

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation
- adopted by most programming languages
- slight modification: only reduce terms not in WHNF

Call-by-Name

- use outermost reduction
- corresponds to lazy evaluation (without memoization)
- e.g., adopted by Haskell
- slight modification: again, only reduce terms not in WHNF

Call-by-Value

- use innermost reduction
- corresponds to strict (or eager) evaluation
- adopted by most programming languages
- slight modification: only reduce terms not in WHNF

Call-by-Name

- use outermost reduction
- corresponds to lazy evaluation (without memoization)
- e.g., adopted by Haskell
- slight modification: again, only reduce terms not in WHNF

Weak Head Normal Form

term t is in weak head normal form iff t is not an application

Abstract Data Types

Idea

- hide implementation details
- just provide interface
- allows to change implementation (e.g., make more efficient) without breaking client code

Idea

- hide implementation details
- just provide interface
- allows to change implementation (e.g., make more efficient) without breaking client code

Haskell

- consider module
```
module M (T, ...) where
type T = C1 |   | CN
```
- only name `T` is exported, but none of `C1` to `CN`
- thus we are not able to directly construct values of type `T`
- if we want to export `C1` to `CN`, we can use `T(..)` in export list

Set Characteristics

- order of elements not important
- no duplicates

Set Characteristics

- order of elements not important
- no duplicates

Examples

$$\{1, 2, 3, 5\} = \{5, 1, 3, 2\}$$
$$\{1, 1, 2, 2\} = \{1, 2\}$$

Set Characteristics

- order of elements not important
- no duplicates

Examples

$$\{1,2,3,5\} = \{5,1,3,2\}$$
$$\{1,1,2,2\} = \{1,2\}$$

Set Operations

description	notation	Haskell
empty set	$\varnothing$	`empty :: Set a`
insertion	$\{x\} \cup S$	`insert :: a -> Set a -> Set a`
membership	$e \in S$	`mem :: a -> Set a -> Bool`
union	$S \cup T$	`union :: Set a -> Set a -> Set a`
difference	$S \setminus T$	`diff :: Set a -> Set a -> Set a`

Example - Sets as Lists

```
module Set (Set,empty,insert,mem,union,diff) where
import qualified Data.List as List
data Set a = Set [a]

empty :: Set a
empty = Set []

insert :: Eq a => a -> Set a -> Set a
insert x (Set xs) = Set (List.nub (x : xs))

mem :: Eq a => a -> Set a -> Bool
mem x (Set xs) = x `elem` xs

union :: Eq a => Set a -> Set a -> Set a
union (Set xs) (Set ys) = Set (List.nub (xs ++ ys))

diff :: Eq a => Set a -> Set a -> Set a
diff (Set xs) (Set ys) = Set (xs List.\\ ys)
```

Note - Imports

- **`import`** `M` imports all functions and types defined in module `M`

Note - Imports

- **`import`** `M` imports all functions and types defined in module `M`
- we may restrict to `f1`, ..., `fN`, writing
  `import` `M (f1,...,fN)`

Note - Imports

- **`import`** `M` imports all functions and types defined in module `M`
- we may restrict to `f1`, . . . , `fN`, writing
  `import` `M (f1,...,fN)`
- by **`import`** `M hiding (f1,...,fN)` we import everything except the functions `f1` to `fN`

Note - Imports

- **`import`** `M` imports all functions and types defined in module `M`
- we may restrict to `f1`, ..., `fN`, writing
  `import` `M (f1,...,fN)`
- by **`import`** `M hiding (f1,...,fN)` we import everything except the functions `f1` to `fN`
- **`import`** `qualified M` allows to access all functions defined in `M` using prefix `M.`

Note - Imports

- **`import`** `M` imports all functions and types defined in module `M`
- we may restrict to `f1`, . . . , `fN`, writing
  `import` `M (f1,...,fN)`
- by **`import`** `M hiding (f1,...,fN)` we import everything except the functions `f1` to `fN`
- **`import`** `qualified M` allows to access all functions defined in `M` using prefix `M.`
- **`import`** `qualified M as N`, same as **`import`** `qualified M` but additionally rename `M` to `N`

New Types

- in `Set` we use **`data`** with a single constructor `Set` to hide the fact that sets are implemented by lists
- this is a common special case
- we may use `newtype Set a = Set a` instead
- only difference: **`newtype`** has better performance than **`data`**

New Types

- in `Set` we use **`data`** with a single constructor `Set` to hide the fact that sets are implemented by lists
- this is a common special case
- we may use **`newtype`** `Set a = Set a` instead
- only difference: **`newtype`** has better performance than **`data`**

Record Syntax

- for data type / new type `T`, instead of `C t1` ... `tN`, we may use
- `C {n1 :: t1, ..., nN :: tN}` as constructor
- provides selector functions `n1::T -> t1`, ..., `nN::T -> tN`

New Types

- in `Set` we use **`data`** with a single constructor `Set` to hide the fact that sets are implemented by lists
- this is a common special case
- we may use **`newtype`** `Set a = Set a` instead
- only difference: **`newtype`** has better performance than **`data`**

Record Syntax

- for data type / new type `T`, instead of `C t1` ... `tN`, we may use
- `C {n1 :: t1, ..., nN :: tN}` as constructor
- provides selector functions `n1::T -> t1`, ..., `nN::T -> tN`

New Types

- in `Set` we use **`data`** with a single constructor `Set` to hide the fact that sets are implemented by lists
- this is a common special case
- we may use **`newtype`** `Set a = Set a` instead
- only difference: **`newtype`** has better performance than **`data`**

Record Syntax

- for data type / new type `T`, instead of `C t1 ... tN`, we may use
- `C {n1 :: t1, ..., nN :: tN}` as constructor
- provides selector functions `n1::T -> t1`, ..., `nN::T -> tN`

Example

- **`data`** `Equation a = E { lhs :: a, rhs :: a }`
- ```
  ghci> let e1 = E "10" "5+5"
  ghci> let e2 = E { rhs = "5+5", lhs = "10" }
  ghci> lhs e1
  "10"
  ghci> rhs e2
  "5+5"
  ```
  ```

Sets and Binary Search Trees

The Type

- we want to use type BTree without prefix

```
import BTree (BTree)
```

- all other functions from BTree with prefix

```
import qualified BTree
```

- the internal representation of a set is a binary tree

```
newtype Set a = Set { rep :: BTree a }
```

The Type

- we want to use type `BTree` without prefix

```
import BTree (BTree)
```

- all other functions from `BTree` with prefix

```
import qualified BTree
```

- the internal representation of a set is a binary tree

```
newtype Set a = Set { rep :: BTree a }
```

Note

- `newtype Set a = Set { rep :: BTree a }` is almost the same as writing `type Set a = BTree a`
- additionally the type system prevents us from "accidentally" (i.e., without the constructor `Set`) using `BTree`s as `Set`s
- no runtime penalty (in contrast to `data Set a = Set { rep :: BTree })`)
- reason: `newtype` restricted to single constructor (usually of same name as newly introduced type),
- whereas `data` may define arbitrary many constructors (e.g., `Empty` and `Node`)

Empty Set

```
empty :: Set a
empty = Set BTree.Empty
```

Empty Set

```
empty :: Set a
empty = Set BTree.Empty
```

Membership

```
mem :: Ord a => a -> Set a -> Bool
mem x s = x `memTree` (rep s)
  where memTree x Empty = False
        memTree x (Node y l r) =
          case compare x y of
            EQ -> True
            LT -> x `memTree` l
            GT -> x `memTree` r
```

Insertion

```haskell
insert :: Ord a => a -> Set a -> Set a
insert x s = Set (insertTree x (rep s))

insertTree :: Ord a => a -> BTree a -> BTree a
insertTree x Empty          = Node x Empty Empty
insertTree x (Node y l r) =
  case compare x y of
    EQ -> Node y l r
    LT -> Node y (insertTree x l) r
    GT -> Node y l (insertTree x r)
```

Union

```haskell
union :: Ord a => Set a -> Set a -> Set a
union s t = Set (rep s `unionTree` rep t)

unionTree :: Ord a => BTree a -> BTree a -> BTree a
unionTree Empty s         = s
unionTree (Node x l r) s =
  insertTree x (l `unionTree` r `unionTree` s)
```

Removing the Maximal Element

```
splitMaxTree :: BTree a -> Maybe (a,BTree a)
splitMaxTree Empty              = Nothing
splitMaxTree (Node x l Empty) = Just (x,l)
splitMaxTree (Node x l r)       =
  let Just (m,r') = splitMaxTree r
  in Just (m,Node x l r')
```

Removing the Maximal Element

```
splitMaxTree :: BTree a -> Maybe (a,BTree a)
splitMaxTree Empty              = Nothing
splitMaxTree (Node x l Empty) = Just (x,l)
splitMaxTree (Node x l r)     =
  let Just (m,r') = splitMaxTree r
  in Just (m,Node x l r')
```

The Maybe Type

- Prelude: `data Maybe a = Just a | Nothing`
- used for type-safe error handling
- if an error occurs, we return `Nothing`
- otherwise `Just` the result

Removing the Maximal Element

```
splitMaxTree :: BTree a -> Maybe (a,BTree a)
splitMaxTree Empty                = Nothing
splitMaxTree (Node x l Empty) = Just (x,l)
splitMaxTree (Node x l r)     =
  let Just (m,r') = splitMaxTree r
  in Just (m,Node x l r')
```

The Maybe Type

- Prelude: `data Maybe a = Just a | Nothing`
- used for type-safe error handling
- if an error occurs, we return `Nothing`
- otherwise `Just` the result

Example - Safe Head

```
safeHead (x:_) = Just x
safeHead _     = Nothing
```

Remove Given Element

```
removeTree :: Ord a => a -> BTree a -> BTree a
removeTree x Empty          = Empty
removeTree x (Node y l r) = case compare x y of
  LT -> Node y (removeTree x l) r
  GT -> Node y l (removeTree x r)
  EQ -> case splitMaxTree l of
    Nothing      -> r
    Just (m,l') -> Node m l' r
```

Remove Given Element

```
removeTree :: Ord a => a -> BTree a -> BTree a
removeTree x Empty          = Empty
removeTree x (Node y l r) = case compare x y of
  LT -> Node y (removeTree x l) r
  GT -> Node y l (removeTree x r)
  EQ -> case splitMaxTree l of
    Nothing     -> r
    Just (m,l') -> Node m l' r
```

Idea

- have binary search tree (BST)
- `x` smaller `y`: `x` can only occur in `l`
- `x` greater `y`: `x` can only occur in `r`
- `x` equals `y`: remove current node and
- combine `l` and `r` into new BST
- therefore, take maximum of `l` as new root
- guarantees that all other elements in `l` are smaller and
- that all elements in `r` are greater

Difference

```
diff :: Ord a => Set a -> Set a -> Set a
diff s t = Set (rep s `diffTree` rep t)

diffTree :: Ord a => BTree a -> BTree a -> BTree a
diffTree t Empty          = t
diffTree t (Node x l r) =
  removeTree x t `diffTree` l `diffTree` r
```

Exercises (for November 19th)

1. Read chapter 3 of Real World Haskell and the lecture notes about the lambda-calculus.
2. Reduce each of the following λ-terms to NF

$$
\begin{gathered}
(\lambda w.\, w)\ ((\lambda xy.\, y)\ (z\ z)) \\
(\lambda xy.\, x)\ (\lambda z.\, y\ z) \\
\lambda z.\, (\lambda x.\, x\ z\ y)\ (\lambda xy.\, y\ z) \\
\lambda xy.\, y\ (\lambda w.\, w)\ (\lambda yz.\, y\ x)
\end{gathered}
$$

3. Reduce ADD 3 2 to WHNF using leftmost innermost/outermost reduction.
4. Give λ-terms encoding `(&&)`, `(||)`, and `not`.
5. Implement safe versions (i.e., using `Maybe`) of `tail`, `init`, and `last`.
6. Implement the function `equals :: Ord a => Set a -> Set a -> Bool`, checking whether two sets are equal.